Massively Parallel Jitter Measurement from Deep Memory Digital Waveforms

Torbjorn Loken*, Lee Barford†*, Frederick C. Harris Jr.*

*Department of Computer Science and Engineering
University of Nevada/0171
Reno, NV 89577-0171 USA

†Measurement Research Laboratory
Agilent Technologies
561 Keystone Ave. Unit 434
Reno, NV, 89503 USA

***Abstract*—As the increase of computer clock rates has finally begun to reach its stopping point the focus has instead shifted to increasing the number of cores present. This increase in cores is seen most in Graphics Processing Units (GPUs) and other massively parallel processors. These new devices however bring with them new methods for achieving top performance. This paper uses jitter quantification to show the power that a massively parallel device can bring to measurement analysis. The parallel algorithm shown is able to obtain large speed ups over serial implementations and is able to analyze large data sets to provide a more complete picture of nature of the timing jitter in a signal.**

Keywords – signal analysis, timing jitter, time interval error, parallel programming, clock recovery

I. Introduction

Measuring jitter is a major consideration when testing any serial, digital communications channel. With requirements for much reduced bit error rates (BERs), the allowable margin for jitter decreases and the need accurately to measure jitter increases. Due to this requirement of low jitter to decrease the bit error rate, tools both quantification and visualization of jitter statistics of a signal are increasingly valuable to digital design engineers. Jitter histograms and eye diagrams are used to not only predict bit error rates, but also give insight to the sources of jitter. As bit error rates decrease, it is desirable to capture longer waveforms (often called "deep waveform") in order to both (1) increase the likelihood of capturing rare events and (2) improve the statistical significance of the tails of the measured jitter histogram or fitted probability distribution used to extrapolate jitter performance and BER [1]. Oscilloscopes are commercially available that can capture billions of samples per trigger. However, clock recovery from the waveform and computing jitter statistics and graphical presentations of the jitter probability distribution becomes a rather computationally expensive task as waveform lengths increase. The purpose of this paper is to present a method for fixed frequency clock recovery and jitter measurement of a two-state digital signal using massive parallel processors and large data sets. The use of parallel processing to analyze signals in the this manner has already been shown to be a strong argument for their for their use for further analysis, like jitter histogram plots and eye diagrams.

The presented method makes minimal assumptions about the signal. The first assumption is that the signal is a two state digital signal. The second is that there is an approximately equal number of low to high and high to low digital state transitions. The implementation of the method for this paper is not IEEE Standard 181-2011 [2] compliant. Importantly the method presented in this paper does not make any assumptions about prior knowledge of the relative high low and middle values of the voltage for the signal or the expected unit interval of the signal.

To facilitate an rapid analysis of a measured waveform, the method presented uses a massively parallel processor. Formerly, massively parallel computation was only available as supercomputers or computing clusters. Both the expense and the physical remoteness of such computing resources made them poor choices for use in measurement systems. But now massively parallel computation is available in the form of Graphics Processor Units (GPUs). GPUs are available that plug into standard PCs at low cost, making them practical for use within measurement systems.

GPUs however are not suited for all problems. Their model of computing pairs extremely well with data parallel tasks and tasks which utilize linear memory access patterns [3]. The key and novel result of this paper is that it is possible to do high throughput clock recovery and jitter statistics computation using a GPU by factoring those computations into a series of steps, each one of which has memory access patterns well suited to the GPU.

We use the Thrust library provided by NVidia [4] gives high level access to programming primitives which are executed on a GPU. These primitives often times have a counter part in the C++ Standard Template Library (STL) [5]. The method proposed makes heavy use of both the iteration and container primitives provided by the Thrust library. This allowed the design of both the CPU and GPU implementations to take place in parallel. The use of Thrust much like the use of the

STL allows for rapid development by combining these simple primitives to implement solutions complex algorithms. Thrust makes heavy use of templating to produce code which is then run on the GPU itself.

A well studied example of a problem which is suited to GPUs is sorting [6]. Sorting algorithms for traditional processes have memory access patterns that are random. However, GPU-specific sorting algorithms have been developed that have linear memory access patterns and extremely high performance on GPUs. Steps of our clock recovery algorithm that might seem to require random memory access patterns are reduced to sorting both (1) to take advantage of the high speed of sorting on the GPU and (2) eliminate the need for random memory access patterns.

The analysis of jitter historically has been largely considered when designing and implementing clock recovery circuits [7]. As bit error rate requirements begin to tighten, it is desirable to identify events with ever more low probability. Therefore, it is becoming increasingly useful to measure jitter and display eye diagrams from much longer waveforms.

Other previous studies of jitter analysis have relied on a reference clock for computing jitter histograms [8]. This is limiting however because a reference clock is not always available in many situations. Due to this limitation approaches which use reference clocks when constructing a jitter histogram are not as generally useful as a method which can make due without.

Recently massively parallel processors have been used in transition localization [9][10]. These methods showed that large speedups of processing time for deep waveforms could be gained for this important first step of many different signal analysis methods. This was an important indicator that further analysis of deep waveforms could be successfully implemented using massively parallel processors.

This paper will explore a method which uses Thrust to give a large speed up to an existing serial algorithm. Section II will discuss the algorithm and its implementation, the results of using the implementation with actual measured data will then be discussed.

II. The Method

This section describes the structure of the method presented in the paper. It will then discuss the changes which were made to maximize the performance on the GPU. Following that is an in depth explanation of the process presented.

The purposed method makes some assumptions about the input signal. Other than the assumption that the signal is digital, the most important of which is that transitions will occur in the signal. Due to this transmission protocols like 8b/10b [11] should be used so that transitions can be guaranteed to happen.

The purposed method is a series of transformations and reductions of the input signal. The first an most important transformation which is preformed is taking the input signal and finding each of the transitions in the signal with sub-sample accuracy. This initial transformation forms the basis of the method. This is accomplished by first determining relative values for the high, low and middle voltages of the input signal. Using these values the transition points in the signal can be found. Once the transition points are found a rough estimate of the number of samples per unit interval is made. Once the rough estimate of the duration of a unit interval is made it is further refined. This refinement is done by comparing the number of samples to the estimated total number of unit intervals in the input signal. With this refined estimate a phase offset estimate can be calculated for each of the transition points in the input signal. Using linear regression the phase offset and a correction for our final estimate can be calculated. From this further analysis tools like jitter histograms or eye diagrams can be constructed.

While the proposed method is straight forward in its design there are considerations which were made to translate the algorithm to work efficiently on the GPU. The most major change is that while the STL provides a primitive for selecting the nth element of a container, Thrust provides no matching call. For the CPU implementation the nth_element was used to obtain percentile data without the overhead of constructing sorted data or histograms. However since this functionality is missing Thrust it was necessary to sort the data each time a percentile was needed. However this is not an issue since Thrust provides a very fast sort [12]. However, the sort destroys its input data, so it was important to maintain proper copies of data which was to be sorted. Another concern when working on the GPU is minimizing the total time spent transferring data over the PCI bus to the card. Care was also taken to reduce all data transfers to the bare minimum, especially in the case of the large data sets used.

A more in depth explanation of the process follows.

Step One: Determine relative high, low and middle voltage levels in the signal. The first step is determining based on the input signal what the values for high and low will be. This done by finding the 1st percentile of the voltages to find the value for low and finding the 99th percentile for the high value. From these values a middle value can be calculated by finding the midpoint of the two values.

Step Two: Find transition points in the signal with in sub-sample accuracy. The second step is finding the all the points of voltage transition in the signal. When a transition is found its position is noted. For each of these transition points a linear interpolation step is done to find the sub-sample transition time. These values are then summed to find each transition point in the signal with sub-sample accuracy.

Step Three: Make a rough estimate of the number of samples in the signal per unit interval. From these transition times a histogram of the inter-transition intervals is made. To make this histogram the inter-transition values are found by finding the adjacent difference of the sub-sample transition times. Using these inter-transition values a histogram is made. The rough estimate of the samples per unit interval is obtained by finding the 25th percentile of these inter-transition times. Using the 25th percentile of these inter-transition times is only valid due to the nature of the signal.

Step Four: Refine the rough estimate of samples per unit

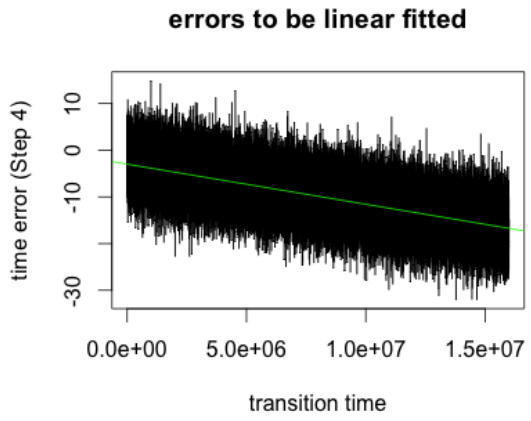

Fig. 1. The estimated time interval error before correction

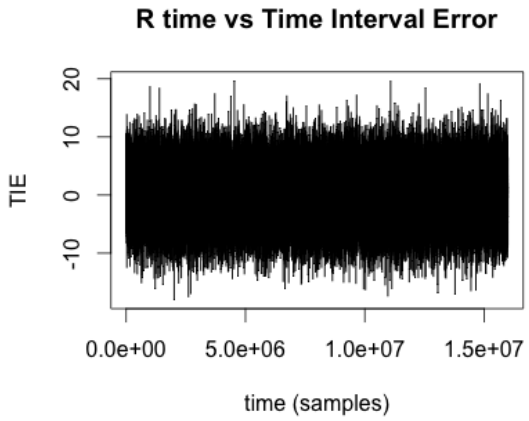

Fig. 2. An example of the final time interval error produced

interval. The rough estimate from the last step will will not be accurate enough for further use. Due to this the rough estimate of samples per unit interval is refined. To do this the first thing needed is the number of estimated unit intervals in the signal. Following that the total number of samples which occur from the first transition to the last transition in the signal. That number of samples is then divided by the estimated number of unit intervals in the signal.

Step Five: Eliminate remaining linear trend in time interval error. In this step the estimate will be corrected a final time using linear regression. Figure 1 shows an example of the linear trend being corrected. In addition to giving the final estimate of the samples per clock, this linear regression step also gives the phase correction need for clock recovery. To correct the refined estimate a linear regression of estimated time interval error given the period estimate obtained in step 4 and the sub-sample transition times from step two. From the linear least squares fitting a coefficient and an offset are obtained. The coefficient is used to correct our refined period estimate from the last step and the offset is the phase offset. Figure 2 shows the result of this step.

Step Six: Construct appropriate plots Once the phase offset is calculated and the estimate of the unit interval has been corrected the appropriate plots are constructed.

III. Results

The proposed algorithm was implemented two times. One implementation used only the CPU and was written in C++ using the primitives provided by the STL to verify the results of the purposed method. The other implementation written in C++ using Thrust to run code on an NVIDIA GPU was used to measure the total speed up of the method with increased core counts. Both implementations were tested with an actual measured waveforms obtained using the apparatus shown in Figure 3, in which waveforms were produced by an Agilent 8133A Pulse Generator and captured by a Agilent DSA91304A Digital Signal Analyzer.

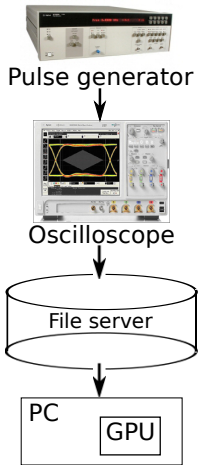

Fig. 3. Measurement apparatus used to verify the proposed method

The experimental platform is an Intel Core2 Quad core CPU Q9450 at 2.66GHz with 8 GB of memory. The GPU used was an NVidia GeForce GTX 480 which contains 15 groupings of 32 compute cores for a total of 480 compute cores and has 1.5 GB of on board memory. The CPU implementation was compiled with g++ version 4.6.3 while due to the restrictions of Thrust the GPU implementation was compiled with the CUDA toolkit version 5.0 and g++ version 4.4.7. All implementation were compiled on 64 bit Ubuntu 12.04 with optimization level -O3.

The measurement results obtained by the serial and GPU versions were identical. For example, when all 16 million samples are used, both versions yield the standard deviation of TIE of 4.630199 samples. The resulting plots of the time interval errors (TIEs) for each transition and the jitter (probability density of the TIE) are shown in Figures 2, 4 and 5. (The probability density is a kernel density estimate [13] computed from the TIEs.)

There were three timings which were measured: one timing for the CPU, one timing for the GPU which did not include

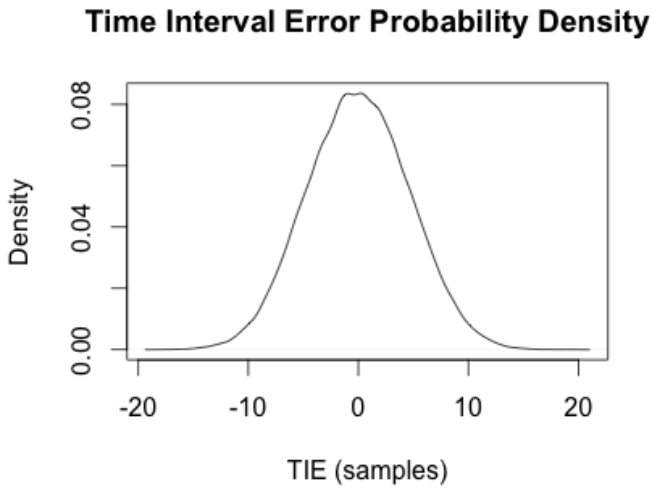

Fig. 4. TIE probability density

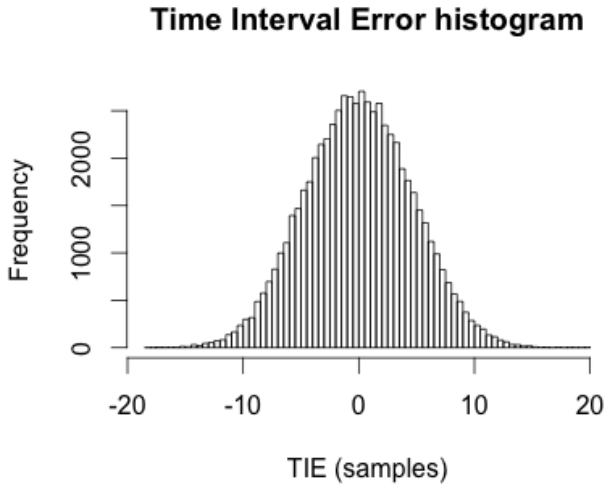

Fig. 5. TIE histogram

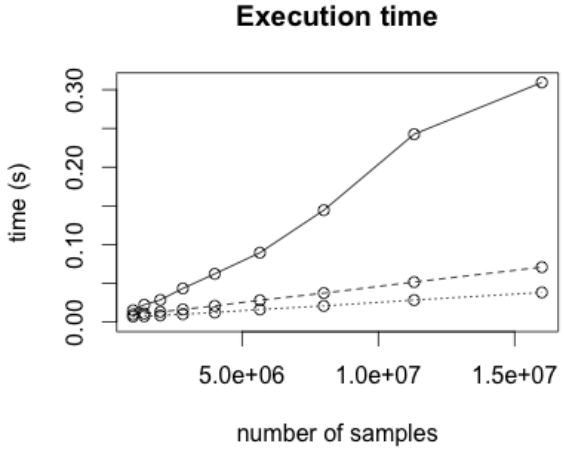

Fig. 6. Execution time of the purposed method in seconds. The solid line shows the CPU execution time, the dashed line shows the GPU execution time including data transfer and the dotted line shows the execution time on the GPU.

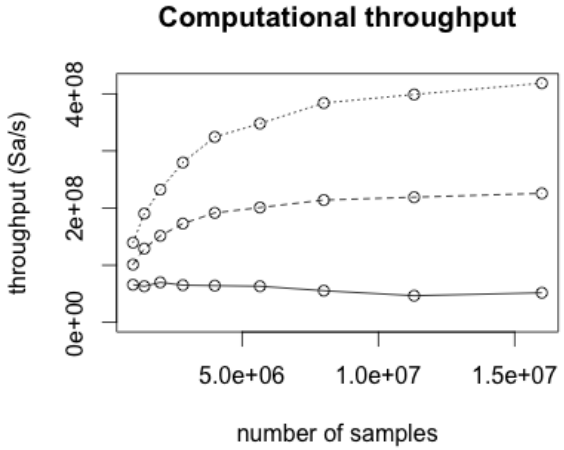

Fig. 7. Computational throughput of the purposed method measured in Samples per second. The solid line shows the throughput of the CPU implementation, the dashed line shows the throughput on the GPU including data transfer and the dotted line shows the throughput on the GPU.

the data transfer times and a one timing for the GPU which included the transfer times as well. Table I and Figure III show the results for these timings as the size of the input signal was increased, while Table II and Figure III shows the through put of these results. For signals less than 16,000,000 samples in length sections of the input waveform were used. The waveforms used in these timing experiments were pseudo-random binary signals (PRBSs).

Table I and Figure III show that when considering only the computations the difference in the serial CPU implementation and the parallel implementation on the GPU as the input size grows is sizable. While the CPU implementation was not expected to keep pace with the GPU implementation it is interesting to note the significant speed up which is present. Another interesting thing to note is that for even with signals as small as 1,000,000 elements, when the data transfer time is considered the GPU implementation is able to handily beat the CPU. This result means that it is quite practical to use this method over a pure CPU implementation. Table II and Figure III show the massive difference in throughput that the GPU is able to provide over the CPU. For the entire 16 million sample signal the GPU implementation was able to handle 8 times as many samples as the CPU in the same amount of time when not considering the total transfer time of data on and off of the GPU. This is a reason expectation to make as the method presented is likely to be an intermediate step in a larger process.

TABLE I
EXECUTION TIME IN MILLISECONDS FOR EACH IMPLEMENTATION FOR EACH OF THE DIFFERENT INPUT SIGNAL LENGTHS

Input Size (samples)	CPU	GPU (no data transfer time)	GPU (with data transfer time)
1,000,000	15.19	7.17	9.89
1,414,213	22.38	7.43	10.96
2,000,000	28.59	8.61	13.21
2,828,427	43.44	10.11	16.36
4,000,000	62.24	12.33	20.88
5,656,854	89.56	16.26	28.15
8,000,000	144.44	20.84	37.38
11,313,708	242.54	28.37	51.66
16,000,000	309.37	38.23	70.88

TABLE II
THROUGHPUT FOR EACH IMPLEMENTATION FOR EACH OF THE DIFFERENT INPUT SIGNAL LENGTHS

Input Size (samples)	CPU (MSa/s)	GPU (MSa/s, no data transfer time)	GPU (MSa/s, with data transfer time)
1,000,000	65.835	139.460	101.122
1,414,213	63.180	190.330	129.046
2,000,000	69.946	232.297	151.401
2,828,427	65.108	279.701	172.917
4,000,000	64.269	324.492	191.561
5,656,854	63.163	347.852	200.963
8,000,000	55.385	383.880	214.013
11,313,708	46.646	398.743	219.002
16,000,000	51.717	418.563	225.740

An interesting thing to note is that despite the relatively high throughput shown by the GPU in Table II and Figure III the application is still computationally bound. While the method as implemented is capable of consuming 418 million samples a second the PCI express bus is capable of transmitting still more samples [14].

IV. FUTURE WORK

To further increase the robustness of the solution provided by this paper we plan on making the implementation comply with the IEEE Standard 181-2011 [2] procedure for (1) identifying voltage reference levels and (2) the location of state transition times. Significant progress toward achieving the second of these on GPUs has been reported elsewhere [9][10] and it should be possible to combine those methods with the present one. The current implementation is non-standard and is more sensitive to noise, glitches, and runt pulses than the procedure specified in the standard. The sensitivity however never presented its self during testing. Ideally this would be done on the GPU to maintain a high level of efficiency by avoiding unnecessary data transfers.

In addition to become more compliant with this standard we plan on experimenting with ways to build a proper histogram on the GPU more efficiently. The current implementation does not need histograms but to become IEEE Standard 181 compliant a future implementation needs an efficient method for building histograms on the GPU.

Finally the presented method is also unable to deal with non-constant clocks. In order to add functionality to recover these non-constant clocks a phase locked loop would be needed.

V. CONCLUSION

The novel contributions of this paper are (1) the application of massively parallel processors to do clock recovery and to quantify jitter and (2) the use of sorting to replace building histograms when analyzing waveforms so as more efficiently to use such processors.

By using the efficient sorting, transformation and reduction routines provided by Thrust, the benefits of constructing a solution using the primitives offered by Thrust were shown. Identical measurement results were obtained, so using parallel processing did not effect the measurement uncertainty. Using Thrust to translate a program written in C++ using the Standard Template Library is a simple process and provides a large speed increase while only requiring a small amount of changes to be made. With this speed comes the possibility of analyzing larger data sets allowing for a more complete measure of the jitter in a signal rather than making many small measurements or attempting to use a statistical method to acquire a similar result. The results of this paper are a strong indication that measurement analysis such as jitter measurements are very well suited for the GPU and that perhaps further optimizations could help alleviate the still present computational bottleneck. Using the purposed method in concert with other GPU powered processes allows for very deep waveforms to be efficiently processed.

REFERENCES

[1] W. Maichen, *Digital Timing Measurement: From Scopes and Probes to Timing and Jitter*. Springer, 2006, ch. 9.3.

[2] "IEEE standard for transitions, pulses, and related waveforms," *IEEE Std 181-2011 (Revision of IEEE Std 181-2003)*, pp. 1 –71, 6 2011.

[3] B. Jang, D. Schaa, P. Mistry, and D. Kaeli, "Exploiting memory access patterns to improve memory performance in data-parallel architectures," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 1, pp. 105–118, 2011.

[4] "Thrust quick start guide," September 2012. [Online]. Available: https://github.com/thrust/thrust/wiki/Quick-Start-Guide

[5] N. M. Josuttis, *The C++ Standard Library: A Tutorial and Reference*, 2nd ed. Addison Wesley, 2012.

[6] E. Sintorn and U. Assarsson, "Fast parallel GPU-sorting using a hybrid algorithm," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1381 – 1388, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731508001196

[7] C. Hogge Jr, "A self correcting clock recovery curcuit," *Lightwave Technology, Journal of*, vol. 3, no. 6, pp. 1312–1314, 1985.

[8] W. Dalal and D. Rosenthal, "Measuring jitter of high speed data channels using undersampling techniques," in *Test Conference, 1998. Proceedings., International*. IEEE, 1998, pp. 814–818.

[9] L. Barford, "Parallelizing small finite state machines, with application to pulsed signal analysis," in *Instrumentation and Measurement Technology Conference (I2MTC), 2012 IEEE International*. IEEE, 2012, pp. 1957–1962.

[10] V. Khambadkar, L. Barford, and F. Harris, "Massively parallel localization of pulsed signal transitions using a GPU," in *Instrumentation and Measurement Technology Conference (I2MTC), 2012 IEEE International*. IEEE, 2012, pp. 2173–2177.

[11] A. X. Widmer and P. A. Franaszek, "A DC-balanced, partitioned-block, 8b/10b transmission code," *IBM Journal of research and development*, vol. 27, no. 5, pp. 440–451, 1983.

[12] N. Leischner, V. Osipov, and P. Sanders, "GPU sample sort," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–10.

[13] B. Silverman, *Density estimation for statistics and data analysis*. Chapman & Hall/CRC, 1986, vol. 26.

[14] M. Koop, W. Huang, K. Gopalakrishnan, and D. Panda, "Performance analysis and evaluation of PCIe 2.0 and quad-data rate infiniband," in *High Performance Interconnects, 2008. HOTI '08. 16th IEEE Symposium on*, Aug. 2008, pp. 85 –92.